A/B Testing - Intermediate Handout

Machine Learning for Smarter Innovation

1 A/B Testing - Intermediate Handout

Target Audience: Practitioners with Python and statistics knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.stats import beta, norm
import statsmodels.stats.proportion as smp
from statsmodels.stats.power import zt_ind_solve_power
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# Reproducibility
np.random.seed(42)

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers practical implementation of A/B testing systems including sample size calculation, frequentist and Bayesian analysis, multi-armed bandits, and sequential testing. A/B testing enables data-driven decisions about product changes by measuring causal effects. The techniques apply across conversion optimization, pricing experiments, feature launches, and algorithm comparison.

1.2 1. Sample Size Calculation

1.2.1 Concept Overview

Proper sample size planning is essential before running any experiment. Underpowered tests waste resources by failing to detect real effects, while overpowered tests delay decisions unnecessarily. Sample size depends on baseline conversion rate, minimum detectable effect (MDE), significance level (alpha), and statistical power. The MDE should reflect the smallest effect worth detecting from a business perspective.

1.2.2 Implementation: Power Analysis

```
def calculate_sample_size(baseline, mde_relative, alpha=0.05, power=0.8):
    """
    Calculate required sample size per group.

    Parameters:
    -----------
    baseline : float
        Current conversion rate (e.g., 0.05 for 5%)
    mde_relative : float
        Minimum detectable effect as relative change (e.g., 0.10 for 10% lift)
    alpha : float
        Significance level (default 0.05)
    power : float
        Statistical power (default 0.8)

    Returns:
    --------
    n : int
        Required sample size per group
    """
    treatment = baseline * (1 + mde_relative)
    effect_size = (treatment - baseline) / np.sqrt(baseline * (1 - baseline))

    n = zt_ind_solve_power(
        effect_size=effect_size,
        alpha=alpha,
        power=power,
        ratio=1.0,
        alternative='two-sided'
    )

    return int(np.ceil(n))


def experiment_duration(n_per_group, daily_traffic, traffic_fraction=0.5):
    """Calculate experiment duration in days."""
    total_needed = 2 * n_per_group
    daily_experiment_traffic = daily_traffic * traffic_fraction
    return total_needed / daily_experiment_traffic


# Example: 5% baseline, want to detect 20% relative lift
baseline = 0.05
mde = 0.20  # 20% relative lift = 6% vs 5%

n_required = calculate_sample_size(baseline, mde)
duration = experiment_duration(n_required, daily_traffic=10000)

print(f"Required sample size per group: {n_required:,}")
print(f"Total users needed: {2 * n_required:,}")
print(f"Estimated duration: {duration:.1f} days")
```

1.2.3 Sample Size Sensitivity

```
def sensitivity_table(baseline, mde_values, power_values):
    """Generate sample size table for different MDE and power combinations."""
    results = []
    for mde in mde_values:
```

```
        for power in power_values:
            n = calculate_sample_size(baseline, mde, power=power)
            results.append({
                'MDE': f'{mde:.0%}',
                'Power': f'{power:.0%}',
                'N per group': n
            })
    return pd.DataFrame(results).pivot(index='MDE', columns='Power', values='N
    per group')


# Generate sensitivity table
mde_values = [0.05, 0.10, 0.15, 0.20]
power_values = [0.80, 0.90]

table = sensitivity_table(0.05, mde_values, power_values)
print("Sample Size Sensitivity (5% baseline):")
print(table)
```

1.3 2. Frequentist A/B Testing

1.3.1 Concept Overview

The frequentist approach tests the null hypothesis that treatment and control have equal conversion rates. The z-test for proportions compares observed conversion rates, producing a p-value and confidence interval. A p-value below alpha (typically 0.05) indicates statistical significance. The confidence interval for the difference shows the range of plausible effect sizes.

1.3.2 Implementation: Proportion Z-Test

```
def proportion_ztest(control, treatment, alpha=0.05):
    """
    Two-proportion z-test for A/B testing.

    Parameters:
    -----------
    control : array-like
        Binary outcomes (0/1) for control group
    treatment : array-like
        Binary outcomes (0/1) for treatment group
    alpha : float
        Significance level

    Returns:
    --------
    results : dict
        Test statistics and interpretation
    """
    n_c, n_t = len(control), len(treatment)
    conv_c, conv_t = sum(control), sum(treatment)
    p_c, p_t = conv_c / n_c, conv_t / n_t

    # Z-test
    count = np.array([conv_t, conv_c])
    nobs = np.array([n_t, n_c])
    z_stat, p_value = smp.proportions_ztest(count, nobs)
```

```
    # Confidence interval for difference
    diff = p_t - p_c
    se_diff = np.sqrt(p_c * (1 - p_c) / n_c + p_t * (1 - p_t) / n_t)
    ci_lower = diff - norm.ppf(1 - alpha/2) * se_diff
    ci_upper = diff + norm.ppf(1 - alpha/2) * se_diff

    # Relative lift
    lift = diff / p_c if p_c > 0 else np.inf

    return {
        'p_control': p_c,
        'p_treatment': p_t,
        'absolute_diff': diff,
        'relative_lift': lift,
        'z_statistic': z_stat,
        'p_value': p_value,
        'ci_lower': ci_lower,
        'ci_upper': ci_upper,
        'significant': p_value < alpha
    }


# Example A/B test
control = np.random.binomial(1, 0.05, 10000)
treatment = np.random.binomial(1, 0.06, 10000)

results = proportion_ztest(control, treatment)

print("Frequentist A/B Test Results:")
print(f"Control: {results['p_control']:.2%}")
print(f"Treatment: {results['p_treatment']:.2%}")
print(f"Absolute diff: {results['absolute_diff']:.2%}")
print(f"Relative lift: {results['relative_lift']:.1%}")
print(f"P-value: {results['p_value']:.4f}")
print(f"95% CI: [{results['ci_lower']:.2%}, {results['ci_upper']:.2%}]")
print(f"Significant: {results['significant']}")
```

1.3.3 T-Test for Continuous Metrics

```
def ttest_ab(control, treatment, alpha=0.05):
    """Welch's t-test for continuous metrics like revenue."""
    mean_c = np.mean(control)
    mean_t = np.mean(treatment)

    t_stat, p_value = stats.ttest_ind(treatment, control, equal_var=False)

    # Confidence interval
    n_c, n_t = len(control), len(treatment)
    var_c, var_t = np.var(control, ddof=1), np.var(treatment, ddof=1)
    se_diff = np.sqrt(var_c / n_c + var_t / n_t)
    diff = mean_t - mean_c

    df = (var_c / n_c + var_t / n_t)**2 / (
        (var_c / n_c)**2 / (n_c - 1) + (var_t / n_t)**2 / (n_t - 1)
    )
    t_crit = stats.t.ppf(1 - alpha/2, df)

    return {
        'mean_control': mean_c,
        'mean_treatment': mean_t,
        'absolute_diff': diff,
```

```
        'relative_lift': diff / mean_c if mean_c > 0 else np.inf,
        't_statistic': t_stat,
        'p_value': p_value,
        'ci_lower': diff - t_crit * se_diff,
        'ci_upper': diff + t_crit * se_diff,
        'significant': p_value < alpha
    }


# Example: Revenue per user
control_rev = np.random.gamma(2, 10, 10000)  # Mean ~$20
treatment_rev = np.random.gamma(2.2, 10, 10000)  # Mean ~$22

rev_results = ttest_ab(control_rev, treatment_rev)
print(f"\nRevenue Test: ${rev_results['mean_treatment']:.2f} vs ${rev_results
    ['mean_control']:.2f}")
print(f"Lift: {rev_results['relative_lift']:.1%}, p-value: {rev_results['
    p_value']:.4f}")
```

1.4 3. Bayesian A/B Testing

1.4.1 Concept Overview

Bayesian A/B testing provides probability statements like "There is a 95% probability that treatment is better than control." Using the Beta-Binomial conjugate prior, we update beliefs about conversion rates as data arrives. Expected loss quantifies the cost of making the wrong decision. Bayesian methods naturally support early stopping and incremental analysis.

1.4.2 Implementation: Beta-Binomial Model

```
def bayesian_ab_test(control_conv, control_n, treatment_conv, treatment_n,
                     prior_alpha=1, prior_beta=1, n_samples=100000):
    """
    Bayesian A/B test using Beta-Binomial model.

    Parameters:
    -----------
    control_conv, control_n : int
        Conversions and visitors in control
    treatment_conv, treatment_n : int
        Conversions and visitors in treatment
    prior_alpha, prior_beta : float
        Beta prior parameters (default: uniform)
    n_samples : int
        Monte Carlo samples

    Returns:
    --------
    results : dict
        Posterior probabilities and expected losses
    """
    # Posterior parameters
    post_a_c = prior_alpha + control_conv
    post_b_c = prior_beta + (control_n - control_conv)
    post_a_t = prior_alpha + treatment_conv
    post_b_t = prior_beta + (treatment_n - treatment_conv)
```

```
    # Sample from posteriors
    samples_c = beta.rvs(post_a_c, post_b_c, size=n_samples)
    samples_t = beta.rvs(post_a_t, post_b_t, size=n_samples)

    # P(treatment > control)
    prob_t_better = (samples_t > samples_c).mean()

    # Expected loss
    loss_choose_c = np.maximum(samples_t - samples_c, 0).mean()
    loss_choose_t = np.maximum(samples_c - samples_t, 0).mean()

    # Lift distribution
    lift = (samples_t - samples_c) / samples_c

    return {
        'posterior_mean_control': post_a_c / (post_a_c + post_b_c),
        'posterior_mean_treatment': post_a_t / (post_a_t + post_b_t),
        'prob_treatment_better': prob_t_better,
        'expected_loss_control': loss_choose_c,
        'expected_loss_treatment': loss_choose_t,
        'lift_mean': lift.mean(),
        'lift_95_ci': (np.percentile(lift, 2.5), np.percentile(lift, 97.5)),
        'samples_control': samples_c,
        'samples_treatment': samples_t
    }


# Example
results = bayesian_ab_test(
    control_conv=500, control_n=10000,
    treatment_conv=580, treatment_n=10000
)

print("Bayesian A/B Test Results:")
print(f"Control rate: {results['posterior_mean_control']:.2%}")
print(f"Treatment rate: {results['posterior_mean_treatment']:.2%}")
print(f"P(Treatment > Control): {results['prob_treatment_better']:.1%}")
print(f"Expected lift: {results['lift_mean']:.1%}")
print(f"95% Credible Interval: [{results['lift_95_ci'][0]:.1%}, {results['
    lift_95_ci'][1]:.1%}]")
print(f"Expected loss (choose Control): {results['expected_loss_control']:.4f}
    ")
print(f"Expected loss (choose Treatment): {results['expected_loss_treatment
    ']:.4f}")
```

1.5 4. Multi-Armed Bandits

1.5.1 Concept Overview

Multi-armed bandits balance exploration (learning which variants are best) with exploitation (sending traffic to the current best). Unlike fixed A/B tests that split traffic equally, bandits adaptively allocate more traffic to better-performing variants. Thompson Sampling samples from posterior distributions and plays the arm with the highest sample, naturally balancing exploration and exploitation.

1.5.2 Implementation: Thompson Sampling

```
class ThompsonSamplingBandit:
```

```
    """Thompson Sampling for multi-armed bandit with binary rewards."""

    def __init__(self, n_arms, prior_alpha=1, prior_beta=1):
        self.n_arms = n_arms
        self.alpha = np.full(n_arms, prior_alpha, dtype=float)
        self.beta = np.full(n_arms, prior_beta, dtype=float)
        self.pulls = np.zeros(n_arms)
        self.rewards = np.zeros(n_arms)

    def select_arm(self):
        """Sample from posteriors, return arm with highest sample."""
        samples = [np.random.beta(self.alpha[i], self.beta[i])
                   for i in range(self.n_arms)]
        return np.argmax(samples)

    def update(self, arm, reward):
        """Update posterior after observing reward."""
        self.alpha[arm] += reward
        self.beta[arm] += (1 - reward)
        self.pulls[arm] += 1
        self.rewards[arm] += reward

    def get_statistics(self):
        """Return posterior means and pull counts."""
        return {
            'posterior_means': self.alpha / (self.alpha + self.beta),
            'pulls': self.pulls,
            'empirical_means': self.rewards / (self.pulls + 1e-8)
        }


# Simulate comparing 3 algorithms
true_rates = [0.05, 0.07, 0.055]  # Algorithm 2 is best
n_rounds = 10000

bandit = ThompsonSamplingBandit(n_arms=3)
cumulative_regret = []
best_arm = np.argmax(true_rates)

for t in range(n_rounds):
    arm = bandit.select_arm()
    reward = np.random.binomial(1, true_rates[arm])
    bandit.update(arm, reward)

    regret = true_rates[best_arm] - true_rates[arm]
    cumulative_regret.append(
        regret if t == 0 else cumulative_regret[-1] + regret
    )

stats = bandit.get_statistics()
print("Thompson Sampling Results:")
for i in range(3):
    print(f"Arm {i+1}: {stats['pulls'][i]:.0f} pulls, "
          f"{stats['empirical_means'][i]:.2%} rate")
print(f"\nTotal regret: {cumulative_regret[-1]:.1f}")
print(f"Best arm allocation: {stats['pulls'][best_arm]/n_rounds:.1%}")
```

1.6 5. Sequential Testing

1.6.1 Concept Overview

Standard A/B tests require committing to a fixed sample size. Sequential testing allows looking at results during the experiment with controlled error rates. O'Brien-Fleming boundaries are conservative early (requiring very strong evidence to stop) and approach nominal significance at the planned end. This enables stopping winners early while maintaining overall alpha at 0.05.

1.6.2 Implementation: Sequential Boundaries

```
def obrien_fleming_boundaries(n_looks, alpha=0.05):
    """Calculate O'Brien-Fleming stopping boundaries."""
    looks = np.arange(1, n_looks + 1)
    z_boundaries = norm.ppf(1 - alpha/2) * np.sqrt(n_looks / looks)
    return z_boundaries


def sequential_test(control, treatment, n_looks=5, alpha=0.05):
    """Perform sequential A/B test with interim analyses."""
    boundaries = obrien_fleming_boundaries(n_looks, alpha)
    n_total = len(control)
    look_sizes = [int(n_total * (i + 1) / n_looks) for i in range(n_looks)]

    results = []
    for i, n in enumerate(look_sizes):
        control_subset = control[:n]
        treatment_subset = treatment[:n]

        test_result = proportion_ztest(control_subset, treatment_subset)
        z = abs(test_result['z_statistic'])
        boundary = boundaries[i]

        results.append({
            'look': i + 1,
            'sample_size': n,
            'z_statistic': z,
            'boundary': boundary,
            'stop': z > boundary,
            'p_value': test_result['p_value']
        })

        if z > boundary:
            break

    return pd.DataFrame(results)


# Example sequential test
control = np.random.binomial(1, 0.05, 10000)
treatment = np.random.binomial(1, 0.065, 10000)  # 30% lift

seq_results = sequential_test(control, treatment, n_looks=5)
print("Sequential Test Results:")
print(seq_results.to_string(index=False))
```

1.7 6. Production A/B Testing Pipeline

1.7.1 Concept Overview

Production experimentation requires systematic processes: pre-experiment checks validate feasibility, guardrail metrics catch regressions, segment analysis detects Simpson's paradox, and decision frameworks translate statistics into actions. A complete pipeline prevents common pitfalls and ensures experiment integrity.

1.7.2 Implementation: Complete Pipeline

```
class ABTestPipeline:
    """End-to-end A/B testing pipeline."""

    def __init__(self, alpha=0.05, power=0.8, mde=0.10):
        self.alpha = alpha
        self.power = power
        self.mde = mde

    def pre_test_check(self, baseline, daily_traffic):
        """Validate experiment feasibility."""
        n_required = calculate_sample_size(baseline, self.mde, self.alpha,
    self.power)
        duration = experiment_duration(n_required, daily_traffic)

        return {
            'n_per_group': n_required,
            'duration_days': duration,
            'feasible': duration <= 30
        }

    def analyze(self, control, treatment):
        """Run both frequentist and Bayesian analysis."""
        freq = proportion_ztest(control, treatment, self.alpha)

        bayes = bayesian_ab_test(
            sum(control), len(control),
            sum(treatment), len(treatment)
        )

        return {
            'frequentist': freq,
            'bayesian': bayes
        }

    def check_guardrails(self, guardrail_results):
        """Identify guardrail failures."""
        failures = []
        for metric, result in guardrail_results.items():
            if result['significant'] and result['absolute_diff'] < 0:
                failures.append(metric)
        return failures

    def decide(self, analysis, guardrail_failures):
        """Generate recommendation."""
        if guardrail_failures:
            return f"ROLLBACK: Guardrail failures in {guardrail_failures}"

        freq = analysis['frequentist']
        bayes = analysis['bayesian']
```

```
        if not freq['significant'] and bayes['prob_treatment_better'] < 0.95:
            return "NO DECISION: Insufficient evidence"

        if bayes['prob_treatment_better'] > 0.95 and freq['relative_lift'] >
    0.05:
            return "SHIP: Strong winner"

        if bayes['prob_treatment_better'] > 0.90:
            return "SHIP WITH MONITORING: Moderate winner"

        return "ITERATE: Effect too small"


# Usage example
pipeline = ABTestPipeline(alpha=0.05, power=0.8, mde=0.15)

# Pre-test
pre_check = pipeline.pre_test_check(baseline=0.05, daily_traffic=5000)
print(f"Pre-test: {pre_check['n_per_group']:,} users needed, {pre_check['
    duration_days']:.1f} days")

# Analysis
control = np.random.binomial(1, 0.05, 8000)
treatment = np.random.binomial(1, 0.058, 8000)

analysis = pipeline.analyze(control, treatment)
decision = pipeline.decide(analysis, guardrail_failures=[])

print(f"\nDecision: {decision}")
```

1.8 Common Parameters and Thresholds

Parameter	Typical Value	Notes
Significance (alpha)	0.05	Lower for high-stakes decisions
Power	0.80	0.90 for important tests
MDE	5-20% relative	Based on business impact
Traffic allocation	50/50	Start equal, adjust for bandits
Minimum runtime	1 week	Capture day-of-week effects
Bayesian threshold	95%	P(B > A) for shipping
Expected loss threshold	0.1%	Maximum acceptable loss

1.9 Practice Projects

1. **Conversion Optimization**: Design and analyze an A/B test for a checkout button color change. Calculate sample size, run the experiment, and present results with both frequentist and Bayesian interpretations.

2. **Pricing Experiment**: Build a Thompson Sampling system to test 4 price points. Track cumulative regret and compare against a fixed A/B test allocation.

3. **Sequential Testing Dashboard**: Implement a monitoring dashboard that displays O'Brien-Fleming boundaries and current test status with automatic early stopping alerts.

4. **Experimentation Platform**: Create a complete pipeline that handles pre-registration, randomization, analysis, and decision documentation for multiple concurrent experiments.

1.10 Troubleshooting

[illegible]	Cause	Solution
Test takes too long	MDE too small	Accept larger MDE or increase traffic
Contradictory results	Multiple testing	Apply Bonferroni correction
Winners become losers	Novelty effect	Run test longer, check for time trends
SRM (sample ratio mismatch)	Randomization bug	Investigate assignment logic

() *	() *	() *
0.3	4	Solution
() *	() *	() *
0.3	4	Guardrail failures	Consider rollback or mitigation
() *	() *	() *
0.3	4	Simpson's paradox	Analyze by segment, use stratification
() *	() *	() *
0.3	4	Use variance metrics	Winsorization or percentile metrics

---

1.11 Next Steps

- Read the advanced handout for causal inference and heterogeneous treatment effects
- Implement variance reduction techniques (CUPED, stratification)
- Build automated experiment monitoring and alerting
- Explore multi-metric optimization and experiment interactions

---

A/B testing transforms opinions into evidence. The combination of proper power analysis, rigorous statistics, and systematic decision frameworks enables confident product decisions. The best experiments ask clear questions, gather sufficient data, and translate results into action.